

ATARI

MADMAC

68000 Macro Assembler Reference Manual

version 1.00

MADMAC Version 1.07 release notes, January, 1990.

Using .cargs used to cause subsequent global symbols to disappear from the symbol table. Now it doesn't.

A movem instruction with no register list (i.e. "movem.l (sp)+,") assembled without error; it now reports "missing register list."

MADMAC Version 1.05 release notes, August, 1988.

This version of MadMac replaces version 1.00 and fixes some small bugs:

Symbols beginning with a capital L were not included in the object file output. They should have been (and now are) excluded only in AS68 compatibility mode, to avoid cluttering the output with compiler-generated symbols.

Forward branches declared "short" (such as bra.s, dbra.s, bsr.s) but whose targets were too far away did not cause an assembly-time error; incorrect machine code was silently generated. They now cause assembly-time errors.

Symbols appeared in the object file symbol table in the order they were referenced in the source, not the order they were declared. Now the order more nearly matches the order of declaration (but not perfectly).

The disp(An,Xn.s) addressing mode produced correct machine code, but incorrect output in the listing file. Now the output is correct.

NOTE: Every effort has been made to ensure the accuracy and robustness of this manual and the associated software. However, because Atari Corporation is constantly improving and updating its computer software, it is unable to guarantee the accuracy of printed or duplicated material after the date of publication and disclaims liability for changes, errors or omissions.

ATARI, the Atari logo, MEGA, and ST are trademarks or registered trademarks of Atari Corporation.

Mark Williams is a trademark of Mark Williams Company.

Motorola is a trademark of Motorola Inc.

*Copyright © 1986, 1987, Atari Corporation
All rights reserved.*

Reproduction of this document or the associated software is not allowed without the specific written consent of Atari Corporation.

Printed in USA.

Atari Document number C300341-001 Rev. A.

Contents

Introduction	2
Getting Started	2
The Command Line	4
Using MADMAC	7
Example Command Lines	7
Interactive Mode	7
Things You Should Be Aware Of	8
Forward Branches	8
Notes for AS68 users	9
Notes for Mark Williams C Users	9
Using MADMAC as a Back-End to the Alcyon C Compiler	10
Text File Format	10
Source Format	11
Statements	11
Equates	11
Symbols and Scope	11
Keywords	12
Constants	13
Strings	13
Register Lists	13
Expressions	15
Order of Evaluation	15
Types	15
Unary Operators	16
Binary Operators	16
Special Forms	16
Example Expressions	17
Directives	18
68000 Mnemonics	22
Macros	24
Macro Declaration	24
Parameter Substitution	24
Macro Invocation	25
Example Macros	26
Repeat Blocks	27
6502 Support	28
6502 Addressing Modes	28
6502 Directives	28
6502 Object Code Format	29
Error Messages	30

Introduction

Introduction

This document describes MADMAC, a fast macro assembler for the 68000. MADMAC currently runs on the Atari ST and under 4.2 BSD VAX UNIX. It was written at Atari Corporation by programmers who needed a high performance assembler for their work.

MADMAC is intended to be used by programmers who write mostly in assembly language. It was not originally a back-end to a C compiler, therefore it has creature comfort that are usually neglected in such back-end assemblers. It supports include files, macros, symbols with limited scope, some limited control structures, and other features. MADMAC is also **blindingly fast**, another feature often sadly and obviously missing in today's assemblers.¹

MADMAC is not entirely compatible with the AS68 assembler provided with the original Atari ST Developer's Kit, but most changes are minor and a few minutes with an editor should allow you to assemble your current source files. If you are an AS68 user, before you leap into the unknown please read the section on Notes for AS68 Users.

This manual was typeset with T_EX and the Computer Modern fonts, and it was printed on the Atari SLM-804 laser printer with a MEGA ST. Except for 200 lines of assembly language, the assembler is written entirely in C.

Getting Started

- ⇒ Write protect your distribution disk and make a backup of it *now*. Put the original disk in a safe place.
- The distribution disk contains a file called **README** that you should read. This file contains important news about the contents of the distribution disk and summarizes the most recent changes to the tools.
- Hard disk users can simply copy the executable files to their work or binary directories. People with floppy disks can copy the executables to ramdisks, install the assembler with the `-q` option, or even work right off of the floppies.
- You will need an editor that can produce "normal" format text files. Micro Emacs will work well, as will most other commercial program editors, but not most word processors (such as First Word or Microsoft Write).
- You will probably want to examine or get a listing of the file **"ATARI.S"**. It contains lots of definitions for the Atari ST, including BIOS variables, most BIOS, XBIOS and GEMDOS traps, and line-A equates. We (or you) could split the file up into pieces (a file for line-A equates, a file for hardware and

¹ It processes 30,000 lines a minute on a lightly loaded VAX 11/780; maybe 40,000 on a 520-ST with an SH-204 hard disk. Yet it could be sped up even more with some effort and without resorting to assembly language; C doesn't have to be slow!

BIOS variables and so on), but MADMAC is so fast that it doesn't matter much.

- Read the rest of the manual, *especially* the first two chapters on **The Command Line** and **Using MADMAC**. The distribution disk contains example programs that you can look at, assemble and modify.

The Command Line

The assembler is called "mac.prg." The command line takes the form:

mac [switches] [files ...]

A command line consists of any number of switches followed by the names of files to assemble. A switch is specified with a dash (-) followed immediately by a key character. Key characters are not case-sensitive, so "-d" is the same as "-D". Some switches accept (or require) arguments to immediately follow the key character, with no spaces in between.

Switch order is important. Command lines are processed from left to right in one pass, and switches usually take effect when they are encountered. In general it is best to specify all switches before the names of any input files.

If the command line is entirely empty then MADMAC prints a copyright message and enters an "interactive" mode, prompting for successive command lines with a star (*). An empty command line will exit (See the examples in the chapter on Using MADMAC). After each assembly in interactive mode, the assembler will print a summary of the amount of memory used, the amount of memory left, the number of lines processed, and the number of seconds the assembly took.

Input files are assumed to have the extension ".s"; if a filename has no extension (i.e. no dot) then ".s" will be appended to it. More than one source filename may be specified: the files are assembled into one object file, as if they were concatenated.

MADMAC normally produces object code in "file.o" if "file.s" is the first input filename. If the first input file is a special character device, the output name is noname.o. The -o switch (see below) can be used change the output file name.

Switch	Description
-dname[=value]	Define symbol, with optional value.
-e[file[.err]]	Direct error messages to the specified file.
-fm	Produce Mark Williams format object file.
-fmu	Like "-fm", but move underscores to end of symbol names.
-ipath	Set include-file directory search path.
-l[file[.prn]]	Construct and direct assembly listing to the specified file.
-ofile[o]	Direct object code output to the specified file.
-p	Produce an executable (.prg) output file.
-ps	Produce an executable (.prg) output file with symbols.
-q	Make MADMAC resident in memory (Atari ST only).
-s	Warn about unoptimized long branches.
-u	Assume that all undefined symbols are external.
-v	Verbose mode (print running dialogue).
-yn	Set listing page size to n lines.
-6	"Back end" mode for Alcyon C68.
file[s]	Assemble the specified file.

The switches are described below. A summary of all the switches is given in the table.

-d

The **-d** switch permits symbols to be defined on the command line. The name of the symbol to be defined immediately follows the switch (no spaces). The symbol name may optionally be followed by an equals sign (=) and a decimal number. If no value is specified the symbol's value is zero. The symbol attributes are "defined, not referenced, and absolute". This switch is most useful for enabling conditionally-assembled debugging code on the command line; for example:

```
-dDEBUG -dLoopCount=999 -dDebugLevel=55
```

-e

The **-e** switch causes MADMAC to send error messages to a file, instead of the console. If a filename immediately follows the switch character, error messages are written to the specified file. If no filename is specified, a file is created with the default extension ".err" and with the root name taken from the first input file name (e.g. error messages are written to "file.err" if "file" or "file.s" is the first input file name). If no errors are encountered, then no error listing file is created. Beware! If an assembly produces no errors, any error file from a previous assembly is not removed.

-fm

-fmu

The **-fm** and **-fmu** switches cause MADMAC to generate Mark Williams style object files instead of Alcyon object files. These files may be linked with the Mark Williams linker. The **-fmu** switch causes underscores on the first character of a global symbol name to be moved to the end of the name, as per the Mark Williams C compiler naming convention. That is, "_main" will become "main_" and "__main" will become "_main_".

-i

The **-i** switch allows automatic directory searching for include files. A list of semi-colon separated directory search paths may be mentioned immediately following the switch (with no spaces anywhere). For example:

```
-im;;c:include;c:include\sys
```

will cause the assembler to search the current directory of device M, and the directories include and include\sys on drive C. If **-i** is not specified, and the environment variable "MACPATH" exists, its value is used in the same manner. For example, users of the Mark Williams shell could put the following line in their profile script to achieve the same result as the **-i** example above:

```
setenv MACPATH="m;;c:include;c:include\sys"
```

-l

The **-l** switch causes MADMAC to generate an assembly listing file. If a filename immediately follows the switch character, the listing is written to the specified file. If no filename is specified, then a listing file is created with the default extension ".prn" and with the root name taken from the first input file name (e.g. the listing is written to "file.prn" if "file" or "file.s" is the first input file name).

-o

The **-o** switch causes MADMAC to write object code on the specified file. No default extension is applied to the filename. For historical reasons the filename can also be separated from the switch with a space (e.g. "-o file").

-p

-ps

The **-p** and **-ps** switches cause MADMAC to produce an Atari ST executable file with the default extension of **".prg"**. If there are any external references at the end of the assembly, an error message is emitted and no executable file is generated. The **-p** switch does not write symbols to the executable file. The **-ps** switch includes symbols (Alcyon format) in the executable file.

-q

The **-q** switch is aimed primarily at users of floppy-disk-only systems. It causes MADMAC to install itself in memory, like a RAMdisk. Then the program **m.prg** (which is very short — less than a sector) can be used instead of **mac.prg**, which can take ten or twelve seconds to load.

-s

The **-s** switch causes MADMAC to generate a list of unoptimized forward branches as warning messages. This is used to point out branches that could have been short (e.g. **"bra"** could be **"bra.s"**).

-u

The **-u** switch takes effect at the end of the assembly. It forces all referenced and undefined symbols to be global, exactly as if they had been made global with a **.extern** or **.globl** directive. This can be used if you have a lot of external symbols, and you don't feel like declaring them all external.

-v

The **-v** switch turns on a "verbose" mode in which MADMAC prints out (for example) the names of the files it is currently processing. Verbose mode is automatically entered when MADMAC prompts for input with a star.

-y

The **-y** switch, followed immediately by a decimal number (with no intervening space), sets the number of lines in a page. MADMAC will produce *N* lines before emitting a form-feed. If *N* is missing or less than 10 an error message is generated.

-6

The **-6** switch takes effect when it is mentioned. It allows MADMAC to be used as a back end to the Alcyon C compiler.¹ Note: the assembler will produce code that is typically ten percent larger and ten percent slower than the output of the Alcyon assembler, therefore use of this switch for production code is discouraged.

¹ The **-6** switch is not a compatibility mode for AS68 — it has been carefully tailored to accept the output of the Alcyon C compiler.

Using MADMAC

Let's assemble and link some example programs. These programs are included on the distribution disk in the "EXAMPLES" directory — you should copy them to your work area before continuing. In the following examples we adopt the conventions that the shell prompt is a percent sign (%) and that your input (the stuff you type) is presented in bold face.

If you have been reading carefully, you know that MADMAC can generate an executable file without linking. This is useful for making small, stand alone programs that don't require externals or library routines. For example, the following two commands:

```
% mac example.s
% aln -s example.o
```

could be replaced by the single command:

```
% mac -ps example.s
```

since you don't need the linker for stand-alone object files.

Successive source files named in the command line are concatenated, as in this example, which assembles three files into a single executable, as if they were one big file:

```
% mac -p bugs shift images
```

Of course you can get the same effect by using the `.include` directive, but sometimes it is convenient to do the concatenation from the command line.

Here we have an unbelievably complex command line:

```
% mac -lzorf -y95 -o tmp -ehack -im: -Ddebug=123 -ps example
```

This produces a listing on the file called "zorf.prn" with 95 lines per page, writes the executable code (with symbols) to a file called "tmp.prg," writes an error listing to the file "hack.err," specifies an include-file path that includes the current directory on the drive "M:," defines the symbol "debug" to have the value 123, and assembles the file "example.s." (Take a deep breath — you got all that?)

One last thing. If there are any assembly errors, MADMAC will terminate with an exit code of 1. If the assembly succeeds (no errors, although there may be warnings) the exit code will be 0. This is primarily for use with "make" utilities.

Interactive Mode

If you invoke MADMAC with an empty command line it will print a copyright message and prompt you for more commands with a star (*). This is useful if you are used to working directly from the desktop, or if you want to assemble several files in succession without having to reload the assembler from disk for each assembly.

In interactive mode, the assembler is also in verbose mode (just as if you had specified `-v` on each command line):

```

% mac
-----
MADMAC      Atari Macro Assembler
Copyright 1987 Atari Corporation
Beta version 0.12   Jun 1987 lmd
-----
* -ps example
[Including: example.s]
[Including: atari.s]
[Leaving: atari.s]
[Leaving: example.s]
[Writing executable file: example.prg]
36K used, 3658K left, 850 lines, 2.0 seconds
*

```

You can see that the assembler gave a "blow-by-blow" account of the files it processed, as well as a summary of the assembly's memory usage, the number of lines processed (including macro and repeat-block expansion), and how long the assembly took.

The assembler prompts for another command with the star. At this point you can either type a new command line for the assembler to process, or you can exit by typing control-C or an empty line.

Things You Should Be Aware Of

MADMAC is a *one pass assembler*. This means that it gets all of its work done by reading each source file exactly once and then "back-patching" to fix up forward references. This one-pass nature is usually transparent to the programmer, with the following important exceptions:

- o In listings, the object code for forward references is not shown. Instead, lowercase "xx"s are displayed for each undefined byte, as in the following example:


```

60xx      1:  bra.s  .2      ; forward branch
xxxxxxx   dc.l  .2      ; forward reference
60FE      .2:  bra.s  .2      ; backward reference

```
- o Forward branches (including BSRs) are never optimized to their short forms. To get a short forward branch it is necessary to explicitly use the ".s" suffix in the source code.
- o Error messages may appear at the end of the assembly, referring to earlier source lines that contained undefined symbols.
- o All object code generated must fit in memory. Running out of memory is a fatal error that you must deal with by splitting up your source files, re-sizing or eliminating memory-using programs such as ramdisks and desk accessories, or buying more RAM.

Forward Branches

MADMAC does not optimize forward branches for you, but it will tell you about them if you use the *-s* (short branch) option:

```

% mac -s example.s
"example.s", line 20: warning: unoptimized short branch

```

With the *-e* option you can redirect the error output to a file, and determine by hand (or editor macros) which forward branches are safe to explicitly declare short.

Notes for AS68 Users

MADMAC is not entirely compatible with the Alcyon assembler, AS68. This section outlines the major differences. In practice, we have found that very few changes are necessary to make AS68 source code assemble.

- o A semicolon (;) must be used to introduce a comment, except that a star (*) may be used in the first column. AS68 treated anything following the operand field, preceeded by whitespace, as a comment. (MADMAC treats a star that is not in column 1 as a multiplication operator).
- o Labels require colons (even labels that begin in column 1).
- o Conditional assembly directives are called **if**, **else** and **endif**. AS68 called these **ifne**, **ifeq** (etc.), and **endc**.
- o The tilde (~) character is an operator, and back-quote (`) is an illegal character. AS68 permitted the tilde and back-quote characters in symbols.
- o There are no equivalents to AS68's **org** or **section** directives. AS68's **page** directive has become **eject**. The AS68 **.xdef** and **.xref** directives are not implemented, but **.globl** makes these unnecessary anyway. The directives **.comline**, **mask2**, **idnt** and **opt**, which were unimplemented and ignored in AS68, are not legal in MADMAC.
- o The location counter cannot be manipulated with a statement of the form:

* = expression
- o The **ds** directive is not permitted in the text or data segments (except in -6 mode); an error message is issued. Use **dcb** instead to reserve large blocks of initialized storage.
- o Back-slashes in strings are "electric" characters that are used to escape C-like character codes. Watch out for GEMDOS path names in ASCII constants — you will have to convert them to double-backslashes.

Notes for Mark Williams C Users

MADMAC will generate object code that the Mark Williams C linker, **ld**, will accept. This has been tested only with version 2.0 of the Mark Williams package. Some notable differences between MADMAC and the Mark Williams assembler, **as**, are:

- o MWC permits 16-character symbol names in the object file, and MADMAC supports this;
- o MWC object files can contain more code and data sections than the MADMAC (Alcyon) object code format. MADMAC maps its code sections as follows:

MWC Space	MADMAC Space
shri (shared instruction)	text
prvi (private instruction)	unsupported
bssi (uninitialized instruction)	unsupported
shrd (shared data)	data
prvd (private data)	unsupported
bssd (uninitialized data)	bss
debug information	unsupported
symbols	symbols
absolute	abs, equates, etc.

It is not possible for MADMAC to generate code in the Mark Williams private instruction, private data or uninitialized instruction spaces.

- None of the Mark Williams assembler directives (e.g. ".long" and ".odd") are supported. None of the MWC non-standard addressing modes are supported.
- The Mark Williams debugger, db, does not support the Alcyon-format symbols produced with the -ps switch; it complains about the format of the executable file and aborts.
- MADMAC does not support the method by which the Mark Williams shell passes long command lines to programs. Command lines are truncated to 127 characters.

Using MADMAC as a Back-End to the Alcyon C Compiler

MADMAC can be used in place of the AS68 assembler as a back-end for the Alcyon version 4.14 C compiler. The "-6" switch turns on a mode that warps and perverts MADMAC's ordinary syntax into accepting what the Alcyon compiler dishes out. This can be used in a batch file (for instance) with a line that looks like:

```
mac -6 -o %1.o m:%1
```

(Assuming that device "M:" is where the source was put by compiler phase c168). You should be aware that AS68 generally produces better and faster code than MADMAC, although it is slower to assemble.

Text File Format

For those using editors other than the "Emacs" style ones (Micro-Emacs, Mince, etc.) this section documents the source file format that MADMAC expects.

- Files must contain characters with ASCII values less than 128; it is not permissible to have characters with their high bits set unless those characters are contained in strings (i.e. between single or double quotes) or in comments.
- Lines of text are terminated with carriage-return/line-feed, linefeed alone, or carriage-return alone.
- The file is assumed to end with the last terminated line. If there is text beyond the last line terminator (e.g. control-Z) it is ignored.

Source Format

Statements

A statement may contain up to four fields which are identified by order of appearance and terminating characters. The general form of an assembler statement is:

```
label: operator    operand(s)    ; comment
```

The label and comment fields are optional. An operand field may not appear without an operator field. Operands are separated with commas. Blank lines are legal. If the first character on a line is an asterisk (*) or semicolon (;) then the entire line is a comment. A semicolon anywhere on the line (except in a string) begins a comment field which extends to the end of the line.

The label, if it appears, must be terminated with a single or double colon. If it is terminated with a double colon it is automatically declared global. It is illegal to declare a confined symbol global (see: **Symbols and Scope**).

Equates

A statement may also take one of these special forms:

```
symbol equ  expression
symbol =    expression
symbol ==   expression
symbol set  expression
symbol reg  register list
```

The first two forms are identical; they equate the symbol to the value of an expression, which must be defined (no forward references). The third form, double-equals (==), is just like an equate except that it also makes the symbol global. (As with labels, it is illegal to make a confined equate global.) The fourth form allows a symbol to be set to a value any number of times, like a variable. The last form equates the symbol to a 16-bit register mask specified by a register list. It is possible to equate confined symbols (see: **Symbols and Scope**). For example:

```
cr      equ      13          ; carriage-return
lf      =        10          ; line-feed
DEBUG   ==        1          ; global debug flag
count   set       0          ; variable
count   set      count + 1    ; increment variable
.regs   reg      d3-d7/a3-a6 ; register list
.cr     =        13          ; confined equate
```

Symbols and Scope

Symbols may start with an uppercase or lowercase letter (A-Z a-z), an underscore (_), a question mark (?) or a period (.). Each remaining character may be an upper or lowercase letter, a digit (0-9), an underscore, a dollar sign (\$), or a question mark. (Periods can only begin a symbol, they cannot appear as a symbol

continuation character). Symbols are terminated with a character that is not a symbol continuation character (e.g. a period or comma, whitespace, etc.). Case is significant for user-defined symbols, but not for 68000 mnemonics, assembler directives and register names. Symbols are limited to 100 characters. When symbols are written to the object file they are silently truncated to eight (or sixteen) characters (depending on the object file format) with no check for (or warnings about) collisions.

For example, all of the following symbols are legal and unique:

```
reallyLongSymbolName .reallyLongConfinedSymbolName
a10 ret move dc frog aa6 a9 ????.
.a1 .ret .move .dc .frog .a9 .9 .????
.0 .00 .000 .1 .11 .111 . .
_frog ?zippo? sys$system atari Atari ATARI aTaRi
```

while all of the following symbols are illegal:

```
12days dc.10 dc.z 'quote .right.here
@work hi.there $money$ ~tilde
```

Symbols beginning with a period (.) are *confined*; their scope is between two normal (unconfined) labels. Confined symbols may be labels or equates. It is illegal to make a confined symbol global (with the ".globl" directive, a double colon, or a double equals). Only unconfined labels delimit a confined symbol's scope; equates (of any kind) do not count. For example, all symbols are unique and have unique values in the following:

```
zero:: subq.w #1,d1
      bmi.s .ret
.loop: clr.w (a0)+
      dbra d0,.loop
.ret: rts
FF:: subq.w #1,d1
     bmi.s .99
.loop: move.w #-1,(a0)+
      dbra d0,.loop
.99: rts
```

Confined symbols are useful since the programmer has to be much less inventive about finding small, unique names that also have meaning.

It is legal to define symbols that have the same names as processor mnemonics (such as "move" or "rts") or assembler directives (such as ".even"). Indeed, one should be careful to avoid typographical errors, such as this classic (in 6502 mode):

```
.6502
.org = $8000
```

which equates a confined symbol to a hexadecimal value, rather than setting the location counter, which the .org directive does (without the equals sign).

Keywords

The following names, in all combinations of uppercase and lowercase, are keywords and may not be used as symbols (e.g. labels, equates, or the names of macros):

```
equ set reg
sr ccr pc sp ssp usp
d0 d1 d2 d3 d4 d5 d6 d7
a0 a1 a2 a3 a4 a5 a6 a7
r0 r1 r2 r3 r4 r5 r6 r7
r8 r9 r10 r11 r12 r13 r14 r15
```

Constants

Numbers may be decimal, hexadecimal, octal, binary or concatenated ASCII. The default radix is decimal, and it may not be changed. Decimal numbers are specified with a string of digits (0-9). Hexadecimal numbers are specified with a leading dollar sign (\$) followed by a string of digits and uppercase or lowercase letters (A-F a-f). Octal numbers are specified with a leading at-sign (@) followed by a string of octal digits (0-7). Binary numbers are specified with a leading percent sign (%) followed by a string of binary digits (0-1). Concatenated ASCII constants are specified by enclosing from one to four characters in single or double quotes. For example:

```
1234      decimal
$1234     hexadecimal
@777      octal
%10111    binary
"z"       ASCII
'frog'    ASCII
```

Negative numbers are specified with a unary minus (-). For example:

```
-5678     -0334     -$4e71
-%11011   -'z'      -"WIND"
```

Strings

Strings are contained between double (") or single (') quote marks. Strings may contain non-printable characters by specifying "backslash" escapes, similar to the ones used in the C programming language. MADMAC will generate a warning if a backslash is followed by a character not appearing below:

```
\\      $5c      backslash
\n      $0a      line-feed (newline)
\b      $08      backspace
\t      $09      tab
\r      $0d      carriage-return
\f      $0c      form-feed
\e      $1b      escape
\'      $27      single-quote
\"      $22      double-quote
```

It is possible for strings (but not symbols) to contain characters with their high bits set (i.e. character codes 128...255).

You should be aware that backslash characters are popular in GEMDOS path names, and that you may have to escape backslash characters in your existing source code. For example, to get the file "c:\auto\ahdi.s" you would specify the string "c:\\auto\\ahdi.s".

Register Lists

Register lists are special forms used with the `movem` mnemonic and the `.reg` directive. They are 16-bit values, with bits 0 through 15 corresponding to registers D0 through A7. A register list consists of a series of register names or register ranges separated by slashes. A register range consists of two register names, `Rm` and `Rn`, $m < n$, separated by a dash. For example:

Source Format

register list	value
-----	----
d0-d7/a0-a7	\$FFFF
d2-d7/a0/a3-a5	\$39FC
d0/d1/a0-a3/d7/a6-a7	\$CF83
d0	\$0001
r0-r15	\$FFFF

Register lists and register equates may be used in conjunction with the `movem` mnemonic, as in this example:

```
temps  reg    d0-d2/a0-a2    ; temp registers
keeps  reg    d3-d7/d3-a6    ; registers to preserve
allregs reg    d0-d7/a0-a7    ; all registers
        movem.l $temps,-(sp)  ; these two lines ...
        movem.l d0-d2/a0-a2,-(sp) ; ... are identical
        movem.l $keeps,-(sp)  ; save "keep" registers
        movem.l (sp)+,$keeps  ; restore "keep" registers
```

Expressions

All values are computed with 32-bit 2's complement arithmetic. For boolean operations (such as `if` or `assert`) zero is considered false, and non-zero is considered true.

Expressions are evaluated strictly left-to-right, with no regard for operator precedence.

Thus the expression "`1 + 2 * 3`" evaluates to 9, not 7. However, precedence may be forced with parenthesis (`()`) or square-brackets (`[]`).

Types

Expressions belong to one of three classes: undefined, absolute or relocatable. An expression is undefined if it involves an undefined symbol (e.g. an undeclared symbol, or a forward reference). An expression is absolute if its value will not change when the program is relocated (for instance, the number 0, all labels declared in an `abs` section, and all Atari ST hardware register locations are absolute values). An expression is relocatable if it involves exactly one symbol that is contained in a text, data or BSS section.

Only absolute values may be used with operators other than addition (+) or subtraction (-). It is illegal, for instance, to multiply or divide by a relocatable or undefined value. Subtracting a relocatable value from another relocatable value in the same section results in an absolute value (the distance between them, positive or negative). Adding (or subtracting) an absolute value to or from a relocatable value yields a relocatable value (an offset from the relocatable address).

It is important to realize that relocatable values belong to the sections they are defined in (e.g. text, data or BSS), and it is not permissible to mix and match sections. For example, in this code:

```
line1: dc.l   line2, line1+8
line2: dc.l   line1, line2-8
line3: dc.l   line2-line1, 8
error: dc.l   line1+line2, line2 >> 1, line3/4
```

Line 1 deposits two longwords that point to line 2. Line 2 deposits two longwords that point to line 1. Line 3 deposits two longwords that have the absolute value eight. The fourth line will result in an assembly error, since the expressions (respectively) attempt to add two relocatable values, shift a relocatable value right by one, and divide a relocatable value by four.

The pseudo-symbol "*" (star) has the value that the current section's location counter had at the *beginning* of the current source line. For example, these two statements deposit three pointers to the label "bar":

```
foo:   dc.l   **4
bar:   dc.l   *, *
```

Similarly, the pseudo-symbol "\$" has the value that the current section's location counter has, and it is kept up to date as the assembler deposits information

"across" a line of source code. For example, these two statements deposit four pointers to the label "zip":

```
zip:   dc.l    $+8, $+4
zop:   dc.l    $, $-4
```

Unary Operators

Operator	Description
-	Unary minus (2's complement).
!	Logical (boolean) NOT.
~	Tilde: bitwise not (1's complement).
^^defined symbol	True if symbol has a value.
^^referenced symbol	True if symbol has been referenced.
^^streq string1,string2	True if the strings are equal.
^^macdef macroName	True if the macro is defined.

- o The boolean operators generate the value 1 if the expression is true, and 0 if it is not.
- o A symbol is referenced if it is involved in an expression. A symbol may have any combination of attributes: undefined and unreferenced, defined and unreferenced (i.e. declared but never used), undefined and referenced (in the case of a forward or external reference), or defined and referenced.

Binary Operators

Operator	Description
+ - * /	The usual arithmetic operators.
%	Modulo.
& ^	Bit-wise AND, OR and Exclusive-OR.
<< >>	Bit-wise shift left and shift right.
< <= >= >	Boolean magnitude comparisons.
=	Boolean equality.
<> !=	Boolean inequality.

- o All binary operators have the same precedence: expressions are evaluated strictly left to right.
- o Division or modulo by zero yields an assembly error
- o The "<>" and "!=" operators are synonyms.
- o Note that the modulo operator (%) is also used to introduce binary constants (see: **Constants**). A percent sign should be followed by at least one space if it is meant to be a modulo operator, and is followed by a '0' or '1'.

Special Forms

Special Form	Description
^^date	The current system date (Gemdos format).
^^time	The current system time (Gemdos format).

- o The "^^date" special form expands to the current system date, in Gemdos format. The format is a 16-bit word with bits 0...4 indicating the day of the month (1...31), bits 5...8 indicating the month (1...12), and bits 9...15 indicating the year since 1980, in the range 0...119.

- o The “`^^time`” special form expands to the current system time, in Gemdos format. The format is a 16-bit word with bits 0...4 indicating the current second divided by 2, bits 5...10 indicating the current minute 0...59, and bits 11...15 indicating the current hour 0...23.

Example Expressions

line	address	contents	source code
1	00000000	4480	lab1: neg.l d0
2	00000002	427900000000	lab2: clr.w lab1
3		=00000084	equ1 = 100
4		=00000098	equ2 = equ1 + 50
5	00000008	00000064	dc.l lab1 + equ1
6	0000000C	7FFFFFFE6	dc.l (equ1 + ^equ2) >> 1
7	00000010	0001	dc.w ^^defined equ1
8	00000012	0000	dc.w ^^referenced lab2
9	00000014	00000002	dc.l lab2
10	00000018	0001	dc.w ^^referenced lab2
11	0000001A	0001	dc.w lab1 = (lab2 - 6)

Lines 1 through four here are used to set up the rest of the example. Line 5 deposits a relocatable pointer to the location 100 bytes beyond the label “lab1.” Line 6 is a nonsensical expression that uses the “`^`” and right-shift operators. Line 7 deposits a word of 1 because the symbol “equ1” is defined (in line 3).

Line 8 deposits a word of 0 because the symbol “lab2,” defined in line 2, has not been referenced. But the expression in line 9 references the symbol “lab2,” so line 10 (which is a copy of line 8) deposits a word of 1. Finally, line 11 deposits a word of 1 because the boolean equality operator evaluates to true.

The operators “`^^defined`” and “`^^referenced`” are particularly useful in conditional assembly. For instance, it is possible to automatically include debugging code if the debugging code is referenced, as in:

```

        lea    string,a0          ; A0 -> message
        jsr    debug              ; print a message
        rts                      ; and return
string: dc.b   "Help me, Spock!",0 ; (the message)
        :
        .iif ^^defined debug, .include "debug.s"

```

The jsr statement references the symbol debug. Near the end of the source file, the “`.iif`” statement includes the file “debug.s” if the symbol debug was referenced. In production code, presumably all references to the debug symbol will be removed, and the debug source file will not be included. (We could have as easily made the symbol debug external, instead of including another source file).

Directives

Assembler directives may be any mix of upper- or lowercase. The leading periods are optional, though they are shown here and their use is encouraged. Directives may be preceded by a label; the label is defined before the directive is executed. Some directives accept size suffixes (.b, .s, .w or .l); the default is word (.w) if no size is specified. The .s suffix is identical to .b. Directives relating to the 6502 are described in the chapter on 6502 Support.

.even

If the location counter for the current section is odd, make it even by adding one to it. In text and data sections a zero byte is deposited if necessary.

.assert expression [,expression...]

Assert that the conditions are true (non-zero). If any of the comma-separated expressions evaluates to zero an assembler warning is issued. For example:

```
.assert *-start = $76
.assert stacksize >= $400
```

.bss

.data

.text

Switch to the BSS, data or text segments. Instructions and data may not be assembled into the BSS segment, but symbols may be defined and storage may be reserved with the .ds directive. Each assembly starts out in the text segment.

.abs [location]

Start an absolute section, beginning with the specified location (or zero, if no location is specified). An absolute section is much like BSS, except that locations declared with .ds are based absolute. This directive is useful for declaring structures or hardware locations.

For example, the following equates:

```
VPLANES = 0
VWRAP   = 2
CONTRL  = 4
INTIN   = 8
PTSIN   = 12
```

could be as easily defined as:

```
.abs
VPLANES: ds.w 1
VWRAP:   ds.w 1
CONTRL:  ds.l 1
INTIN:   ds.l 1
PTSIN:   ds.l 1
```

.comm symbol, expression

Specifies a label and the size of a common region. The label is made global, thus confined symbols cannot be made common. The linker groups all common regions of the same name; the largest size determines the real size of the common region when the file is linked.

.dc[.size] expression [,expression...]

Deposit initialized storage in the current section. If the specified size is word or long, the assembler will execute a `.even` before depositing data. If the size is `.b`, then strings that are not part of arithmetic expressions are deposited byte-by-byte. If no size is specified, the default is `.w`. This directive cannot be used in the BSS section.

.dcb[.size] expression1, expression2

Generate an initialized block of `expression1` bytes, words or longwords of the value `expression2`. If the specified size is word or long, the assembler will execute `.even` before generating data. If no size is specified, the default is `.w`. This directive cannot be used in the BSS section.

.ds[.size] expression

Reserve space in the current segment for the appropriate number of bytes, words or longwords. If no size is specified, the default size is `.w`. If the size is word or long, the assembler will execute `.even` before reserving space. This directive can only be used in the BSS or ABS sections (in text or data, use `.dcb` to reserve large chunks of initialized storage.)

.init[.size] [#expression,]expression[.size] [, ...]

Generalized initialization directive. The size specified on the directive becomes the default size for the rest of the line. (The "default" default size is `.w`.) A comma-separated list of expressions follows the directive; an expression may be followed by a size to override the default size. An expression may be preceded by a sharp sign, an expression and a comma, which specifies a repeat count to be applied to the next expression. For example:

```
.init.l -1, 0.w, #16,'z'.b, #3,0, 11.b
```

will deposit a longword of -1, a word of zero, sixteen bytes of lower-case 'z', three longwords of zero, and a byte of 11.

No auto-alignment is performed within the line, but a `.even` is done once (before the first value is deposited) if the default size is word or long.

.cargs [#expression,]symbol[.size] [, symbol[.size]...]

Compute stack offsets to C (and other language) arguments. Each symbol is assigned an absolute value (like `equ`) which starts at `expression` and increases by the size of each symbol, for each symbol. If the `expression` is not supplied, the default starting value is 4. For example:

```
.cargs #8, .fileName.l, .openMode, .bufPointer.l
```

could be used to declare offsets from A6 to a pointer to a filename, a word containing an open mode, and a pointer to a buffer. (Note that the symbols used here are confined). Another example, a C-style "string-length" function, could be written as:

```

_strlen:: .carga .string      ; declare arg
        move.l .string(sp),a0 ; a0 -> string
        moveq #-1,d0          ; initial size = -1
.1:      addq.l #1,d0          ; bump size
        tst.b (a0)+           ; at end of string?
        bne .1                ; (no -- try again)
        rts                   ; return string length

```

.end

End the assembly. In an include file, end the include file and resume assembling the superior file. This statement is not required, nor are warning messages generated if it is missing at the end of a file. This directive may be used inside conditional assembly, macros or **.rept** blocks.

.if expression**.else****.endif**

Start a block of conditional assembly. If the expression is true (non-zero) then assemble the statements between the **.if** and the matching **.endif** or **.else**. If the expression is false, ignore the statements unless a matching **.else** is encountered. Conditional assembly may be nested to any depth.

It is possible to exit a conditional assembly block early from within an include file (with **end**) or a macro (with **endm**).

.iif expression, statement

Immediate version of **.if**. If the expression is true (non-zero) then the statement, which may be an instruction, a directive or a macro, is executed. If the expression is false, the statement is ignored. No **.endif** is required. For example:

```

.iif age < 21, canDrink = 0
.iif weight > 500, dangerFlag = 1
.iif !(defined DEBUG). include dbsrc

```

.macro name [formal, formal, ...]**.endm****.exitm**

Define a macro called *name* with the specified formal arguments. The macro definition is terminated with a **.endm** statement. A macro may be exited early with the **.exitm** directive. See the chapter on **Macros** for more information.

.undefmac macroName [, macroName...]

Remove the macro-definition for the specified macro names. If reference is made to a macro that is not defined, no error message is printed and the name is ignored.

.rept expression**.endr**

The statements between the **.rept** and **.endr** directives will be repeated *expression* times. If the expression is zero or negative, no statements will be assembled. No label may appear on a line containing either of these directives.

.globl symbol [, symbol...]**.extern symbol [, symbol...]**

Each symbol is made global. None of the symbols may be confined symbols (those starting with a period). If the symbol is defined in the assembly, the symbol is exported in the object file. If the symbol is undefined at the end of the assembly, and it was referenced (i.e. used in an expression), then the

symbol value is imported as an external reference that must be resolved by the linker. The `.extern` directive is merely a synonym for `.globl`.

`.include "file"`

Include a file. If the filename is not enclosed in quotes, then a default extension of `".s"` is applied to it. If the filename is quoted, then the name is not changed in any way.

Note: If the filename is not a valid symbol, then the assembler will generate an error message. You should enclose filenames such as `"atari.s"` in quotes, because such names are not symbols.

If the include file cannot be found in the current directory, then the directory search path, as specified by `-i` on the commandline, or by the `MACPATH` environment string, is traversed

`.eject`

Issue a page eject in the listing file.

`.title "string"`

`.subttl [-] "string"`

Set the title or subtitle on the listing page. The title should be specified on the the first line of the source program in order to take effect on the first page. The second and subsequent uses of `.title` will cause page ejects. The second and subsequent uses of `.subttl` will cause page ejects unless the subtitle string is preceeded by a dash (-).

`.list`

`.nlist`

Enable or disable source code listing. These directives increment and decrement an internal counter, so they may be appropriately nested. They have no effect if the `-l` switch is not specified on the commandline.

`.goto label`

This directive provides unstructured flow of control within a macro definition. It will transfer control to the line of the macro containing the specified `goto label`. A `goto label` is a symbol preceeded by a colon that appears in the first column of a source line within a macro definition:

```
: label_
```

where the label itself can be any valid symbol name, followed immediately by whitespace and a valid source line (or end of line). The colon **must** appear in the first column.

The `goto-label` is removed from the source line prior to macro expansion - to all intents and purposes the label is invisible except to the `.goto` directive. Macro expansion does not take place within the label.

For example, here is a silly way to count from 1 to 10 without using `.rept`:

```

        .macro Count
count    set      1
:loop    dc.w     count
count    set      count + 1
        iif count <= 10, goto loop
        .endm

```

68000 Mnemonics

Mnemonics

All of the standard Motorola 68000 mnemonics and addressing modes are supported; you should refer to **The Motorola M68000 Programmer's Reference Manual** for a description of the instruction set and the allowable addressing modes for each instruction. With one major exception (forward branches) the assembler performs all the reasonable optimizations of instructions to their short or address register forms.

Register names may be in upper or lower case. The alternate forms R0 through R15 may be used to specify D0 through A7. All register names are keywords, and may not be used as labels or symbols. None of the 68010 or 68020 register names are keywords (but they may become keywords in the future).

Addressing Modes

Assembler Syntax	Description
Dn	Data register direct
An	Address register direct
(An)	Address register indirect
(An)+	Address register indirect postincrement
-(An)	Address register indirect predecrement
disp(An)	Address register indirect with displacement
bdisp(An, Xi[size])	Address register indirect indexed
abs.w	Absolute short
abs	Absolute (long or short)
abs.l	Forced absolute long
disp(PC)	Program counter with displacement
bdisp(PC, Xi)	Program counter indexed
#imm	Immediate

Branches

Since MADMAC is a one pass assembler, forward branches cannot be automatically optimized to their short form. Instead, unsized forward branches are assumed to be long. Backward branches are always optimized to the short form if possible.

A table that lists "extra" branch mnemonics (common synonyms for the Motorola defined mnemonics) appears below.

Linker Constraints

It is not possible to make an external reference that will fix up a byte. For example:

```
extern frog
move.l frog(pc,d0),d1
```

Branch Synonyms

Alternate name	Becomes:
bhs	bcc
blo	bcs
bze, bz	beq
bns	bne
dblo	dbcs
dbze	dbeq
dbra	dbf
dbhs	dbhi
dbnz	dbne

is illegal (and generates an assembly error) when `frog` is external, because the displacement occupies a byte field in the 68000 offset word, which the object file cannot represent.

Optimizations and Translations

The assembler provides "creature comforts" when it processes 68000 mnemonics:

- o `CLR.x An` will really generate `SUB.x An,An`.
- o `ADD`, `SUB` and `CMP` with an address register will really generate `ADDA`, `SUBA` and `CMPA`.
- o The `ADD`, `AND`, `CMP`, `EOR`, `OR` and `SUB` mnemonics with immediate first operands will generate the "I" forms of their instructions (`ADDI`, etc.) if the second operand is not register direct.
- o All shift instructions with no count value assume a count of one.
- o `MOVE.L` is optimized to `MOVEQ` if the immediate operand is defined and in the range `-128...127`. However, `ADD` and `SUB` are never translated to their quick forms; `ADDQ` and `SUBQ` must be explicit.

Macros

A macro definition is a series of statements of the form:

```
.macro name [ formal-arg, ...]  
:  
    statements making up the macro body  
:  
.endm
```

The name of the macro may be any valid symbol that is not also a 68000 instruction or an assembler directive. (The name may begin with a period — macros cannot be made confined the way labels or equated symbols can be). The formal argument list is optional; it is specified with a comma-separated list of valid symbol names. Note that there is no comma between the name of the macro and the name of the first formal argument.

A macro body begins on the line after the `.macro` directive. All instructions and directives, except other macro definitions, are legal inside the body.

The macro ends with the `.endm` statement. If a label appears on the line with this directive, the label is ignored and a warning is generated.

Parameter Substitution

Within the body, formal parameters may be expanded with the special forms:

```
\name  
\{name}
```

The second form (enclosed in braces) can be used in situations where the characters following the formal parameter name are valid symbol continuation characters. This is usually used to force concatenation, as in:

```
\{frog}star  
\{godzilla}vs\{reagan}
```

The formal parameter name is terminated with a character that is not valid in a symbol (e.g. whitespace or punctuation); optionally, the name may be enclosed in curly-braces. The names must be symbols appearing on the formal argument list, or a single decimal digit (\1 corresponds to the first argument, \2 to the second, \9 to the ninth, and \0 to the tenth). It is possible for a macro to have more than ten formal arguments, but arguments 11 and on must be referenced by name, not by number.

Other special forms are:

```

        .macro foo source
            .iif !\?source, .exitm ; exit if source is empty
            move \source,d0        ; otherwise, deposit source
        .endm

```

will not generate the move instruction if the argument "source" is missing from the macro invocation.

The `.end`, `.endif` and `.exitm` directives all pop-out of their include levels appropriately. That is, if a macro performs a `.include` to include a source file, an executed `.exitm` directive within the include-file will pop out of both the include-file and the macro.

Macros may be recursive or mutually recursive to any level, subject only to the availability of memory. When writing recursive macros, take care in the coding of the termination condition(s). A macro that repeatedly calls itself will cause the assembler to exhaust its memory and abort the assembly.

Example Macros

The `Gemdos` macro is used to make file system calls. It has two parameters, a function number and the number of bytes to clean off the stack after the call. The macro pushes the function number onto the stack and does the trap to the file system. After the trap returns, conditional assembly is used to choose an `addq` or an `add.w` to remove the arguments that were pushed.

```

        .macro Gemdos trpno, clean
            move.w    #\trpno,-(sp) ; push trap number
            trap      #1            ; do GEMDOS trap
            .if \clean <= 8         ;
            addq      #\clean,sp    ; clean-up up to 8 bytes
            .else                  ;
            add.w     #\clean,sp    ; clean-up more than 8 bytes
            .endif
        .endm

```

The `Fopen` macro is supplied two arguments; the address of a filename, and the open mode. Note that plain move instructions are used, and that the caller of the macro must supply an appropriate addressing mode (e.g. immediate) for each argument.

```

        .macro Fopen file, mode
            move.w    \mode,-(sp) ; push open mode
            move.l    \file,-(sp) ; push address of file name
            Gemdos    $3d,8       ; do the GEMDOS call
        .endm

```

The `String` macro is used to allocate storage for a string, and to place the string's address somewhere. The first argument should be a string or other expression acceptable in a `dc.b` directive. The second argument is optional; it specifies where the address of the string should be placed. If the second argument is omitted, the string's address is pushed onto the stack. The string data itself is kept in the data segment.

Special Form	Description
\\	a single "\"
\-	a unique label of the form "Mn"
\#	the number of arguments actually specified
\!	the "dot-size" specified on the macro invocation
\?name	conditional expansion
\?{name}	conditional expansion

The last two forms are identical: if the argument is specified and is non-empty, the form expands to a "1", otherwise (if the argument is missing or empty) the form expands to a "0".

The form "\!" expands to the "dot-size" that was specified when the macro was invoked. This can be used to write macros that behave differently depending on the size suffix they are given, as in this macro which provides a synonym for the "dc" directive:

```
.macro deposit value
dc\!    \value
.endm
deposit.b 1      ; byte of 1
deposit.w 2      ; word of 2
deposit.l 3      ; longword of 3
deposit  4      ; word of 4 (no explicit size)
```

Macro Invocation

A previously-defined macro is called when its name appears in the operation field of a statement. Arguments may be specified following the macro name; each argument is separated by a comma. Arguments may be empty. Arguments are stored for substitution in the macro body in the following manner:

- o Numbers are converted to hexadecimal.
- o All spaces outside strings are removed.
- o Keywords (such as register names, dot sizes and "---" operators) are converted to lowercase.
- o Strings are enclosed in double-quote marks (").

For example, a hypothetical call to the macro "mymacro", of the form:

```
mymacro A0, , 'Zorch' / 32, --DEFINED foo, , , tick tock
```

will result in the translations:

Argument	Expansion	Comment
\1	a0	"A0" converted to lower-case
\2		empty
\3	"Zorch"/\$20	"Zorch" in double-quotes, 32 in hexadecimal
\4	--defined foo	"--DEFINED" converted to lower-case
\5		empty
\6		empty
\7	ticktock	spaces removed (note concatenation)

The .exitm directive will cause an immediate exit from a macro body. Thus the macro definition:

```

        .macro String str,loc
            .if \?loc          ; if loc is defined
                move.l #.\~, \loc ; put the string's address there
            .else              ; otherwise
                pea .\~        ; push the string's address
            .endif
            .data              ; put the string data
            .\~: dc.b \str,0    ; in the data segment
            .text              ; and switch back to the text segment
        .endm

```

The construction “.\~” will expand to a label of the form “.Mn” (where n is a unique number for every macro invocation), which is used to tag the location of the string. The label should be confined because the macro may be used along with other confined symbols.

Unique symbol generation plays an important part in the art of writing fine macros. For instance, if we needed three unique symbols, we might write “.a\~”, “.b\~” and “.c\~”.

Repeat Blocks

Repeat-blocks provide a simple iteration capability. A repeat block allows a range of statements to be repeated a specified number of times. For instance, to generate a table consisting of the numbers 255 through 0 (counting backwards) you could write:

```

        .count set    255          ; initialize counter
        .rept 256          ; repeat 256 times:
            dc.b    .count          ; deposit counter
        .count set    .count - 1    ; and decrement it
        .endr              ; (end of repeat block)

```

Repeat blocks can also be used to duplicate identical pieces of code (which are common in bitmap-graphics routines). For example:

```

        .rept 16          ; clear 16 words
        clr.w    (a0)+      ; starting at A0
        .endr

```

6502 Support

MADMAC will generate code for the Motorola 6502 microprocessor. This chapter describes extra addressing modes and directives used to support the 6502.

As the 6502 object code is not linkable (currently there is no linker) external references may not be made. (Nevertheless, MADMAC may reasonably be used for large assemblies because of its blinding speed.)

All standard 6502 addressing modes are supported, with the exception of the accumulator addressing form, which must be omitted (e.g. "ror a" becomes "ror"). Five extra modes, synonyms for existing ones, are included for compatibility with the Atari Coinop assembler.

<i>empty</i>	implied or accumulator (e.g. <i>tsx</i> or <i>ror</i>)
<i>expr</i>	absolute or zeropage
<i>#expr</i>	immediate
<i>(expr,x)</i>	indirect X
<i>(expr),y</i>	indirect Y
<i>(expr)</i>	indirect
<i>expr,x</i>	indexed X
<i>expr,y</i>	indexed Y
<i>@expr(x)</i>	indirect X
<i>@expr(y)</i>	indirect Y
<i>@expr</i>	indirect
<i>x,expr</i>	indexed X
<i>y,expr</i>	indexed Y

While MADMAC lacks "high" and "low" operators, high bytes of words may be extracted with the shift (>>) or divide (/) operators, and low bytes may be extracted with the bitwise AND (&) operator.

.6502

This directive enters the 6502 section. The location counter is undefined, and must be set with ".org" before any code can be generated.

The "dc.w" directive will produce 6502-format words (low byte first). The 68000's reserved keywords (*d0-d7/a0-a7/ssp/usp* and so on) remain reserved (and thus unusable) while in the 6502 section. The directives *globl*, *dc.l*, *dcb.l*, *text*, *data*, *bss*, *abs*, *even* and *comm* are illegal in the 6502 section. It is permitted, though probably not useful, to generate both 6502 and 68000 code in the same object file.

.68000

This directive leaves the 6502 segment and returns to the 68000's text segment. 68000 instructions may be assembled as normal.

.org location

This directive is only legal in the 6502 section. It sets the value of the location counter (or *pc*) to *location*, an expression that must be defined, absolute, and less than \$10000.

WARNING

It is possible to assemble "beyond" the microprocessor's 64K address space, but attempting to do so will probably screw up the assembler. DO NOT attempt to generate code like this:

```
.org $ffff
nop
nop
nop
```

the third NOP in this example, at location \$10000, may cause the assembler to crash or exhibit spectacular schizophrenia. In any case, MADMAC will give no warning before flaking out.

Object Code Format

This is a little bit of a kludge. An object file consists of a page map, followed by one or more page images, followed by a normal Alcyon 68000 object file. If the page map is all zero, it is not written.

The page map contains a byte for each of the 256 256-byte pages in the 6502's 64K address space. The byte is zero (\$00) if the page contained only zero bytes, or one (\$01) if the page contained any non-zero bytes. If a page is flagged with a one, then it is written (in order) following the page map.

The following code:

```
.6502
.org $8000
.dc.b 1
.org $8100
.dc.b 1
.org $8300
.dc.b 1
.end
```

will generate a page map that looks (to a programmer) something like:

```
<$80 bytes of zero>
01 01 00 01
<$7c more bytes of zero, for $100 total>
<image of page $80>
<image of page $81>
<image of page $83>
```

Following the last page image is an Alcyon-format object file, starting with the magic number \$601a. It may contain 68000 code (although that is probably useless), but the symbol table is valid and available for debugging purposes. 6502 symbols will be absolute (not in text, data or bss).

Error Messages

When Things Go Wrong

Most of MADMAC's error messages are self-explanatory. They fall into four classes: *warnings* about situations that you (or the assembler) may not be happy about, *errors* that cause the assembler to not generate object files, *fatal errors* that cause the assembler to abort immediately, and *internal errors* that should never happen.¹

You can write editor macros (or sed or awk scripts) to parse the error messages MADMAC generates. When a message is printed, it is of the form:

`"filename", line line-number: message`

The first element, a filename enclosed in double quotes, indicates the file that generated the error. The filename is followed by a comma, the word "line", and a line number, and finally a colon and the text of the message. The filename "(*top*)" indicates that the assembler could not determine which file had the problem.

The following sections list warnings, errors and fatal errors in alphabetical order, along with a short description of what may have caused the problem.

Warnings

bad backslash code in string

You tried to follow a backslash in a string with a character that the assembler didn't recognize. Remember that MADMAC uses a C-style escape system in strings.

label ignored

You specified a label before a **macro**, **rept** or **endm** directive. The assembler is warning you that the label will not be defined in the assembly.

unoptimized short branch

This warning is only generated if the **-s** switch is specified on the command line. The message refers to a forward, unsized long branch that you could have made short (**.s**).

Fatal Errors

cannot continue

As a result of previous errors, the assembler cannot continue processing. The assembly is aborted.

line too long as a result of macro expansion

When a source line within a macro was expanded, the resultant line was too long for MADMAC (longer than 200 characters or so).

¹ If you come across an internal error, we would appreciate it if you would contact Atari Technical Support and let us know about the problem.

memory exhausted

The assembler ran out of memory. You should (1) split up your source files and assemble them separately, or (2) if you have any ramdisks or RAM-resident programs (like desk accessories) decrease their size so that the assembler has more RAM to work with. As a rule of thumb, pure 68000 code will use up to twice the number of bytes contained in the source files, whereas 6502 code will use 64K of ram right away, plus the size of the source files. The assembler itself uses about 80K bytes. Get out your calculator...

too many ENDMs

The assembler ran across an `endm` directive when it wasn't expecting to see one. The assembly is aborted. Check the nesting of your macro definitions — you probably have an extra `endm`.

Errors**.cargs syntax**

Syntax error in `.cargs` directive.

.comm symbol already defined

You tried to `.comm` a symbol that was already defined.

.ds permitted only in BSS

You tried to use `.ds` in the text or data section.

.init not permitted in BSS or ABS

You tried to use `.init` in the BSS or ABS section.

.org permitted only in .6502 section

You tried to use `.org` in a 68000 section.

Cannot create: filename

The assembler could not create the indicated filename.

External quick reference

You tried to make the immediate operand of a `moveq`, `subq` or `addq` instruction external.

PC-relative expr across sections

You tried to make a PC-relative reference to a location contained in another section.

[bws] must follow '.' in symbol

You tried to follow a dot in a symbol name with something other than one of the four characters 'B', 'W', 'S' or 'L'.

addressing mode syntax

You made a syntax error in an addressing mode.

assert failure

One of your `.assert` directives failed!

bad (section) expression

You tried to mix and match sections in an expression.

bad 6502 addressing mode

The 6502 mnemonic will not work with the addressing mode you specified.

bad expression

There's a syntax error in the expression you typed.

Error Messages

bad size specified

You tried to use an inappropriate size suffix for the instruction. Check your 68000 manual for allowable sizes.

bad size suffix

You can't use `.b` (byte) mode with the `movem` instruction.

cannot .globl local symbol

You tried to make a confined symbol global or common.

cannot initialize non-storage (BSS) section

You tried to generate instructions (or data, with `dc`) in the BSS or ABS section.

cannot use '.b' with an address register

You tried to use a byte-size suffix with an address register. The 68000 does not perform byte-sized address register operations.

directive illegal in .6502 section

You tried to use a 68000-oriented directive in the 6502 section.

divide by zero

The expression you typed involves a division by zero.

expression out of range

The expression you typed is out of range for its application.

external byte reference

You tried to make a byte-sized reference to an external symbol, which the object file format will not allow

external short branch

You tried to make a short branch to an external symbol, which the linker cannot handle.

extra (unexpected) text found after addressing mode

MADMAC thought it was done processing a line, but it ran up against "extra" stuff. Be sure that any comment on the line begins with a semicolon, and check for dangling commas, etc.

forward or undefined .assert

The expression you typed after a `.assert` directive had an undefined value. Remember that MADMAC is one-pass.

hit EOF without finding matching .endif

The assembler fell off the end of last input file without finding a `.endif` to match an `.if`. You probably forgot a `.endif` somewhere.

illegal 6502 addressing mode

The 6502 instruction you typed doesn't work with the addressing mode you specified.

illegal absolute expression

You can't use an absolute-valued expression here.

illegal bra.s with zero offset

You can't do a short branch to the very next instruction (read your 68000 manual).

illegal byte-sized relative reference

The object file format does not permit bytes contain relocatable values; you tried to use a byte-sized relocatable expression in an immediate addressing mode.

illegal character

Your source file contains a character that MADMAC doesn't allow. (most control characters fall into this category).

illegal initialization of section

You tried to use .dc or .dcb in the BSS or ABS sections.

illegal relative address

The relative address you specified is illegal because it belongs to a different section.

illegal word relocatable (in .PRG mode)

You can't have anything other than long relocatable values when you're generating a .PRG file.

inappropriate addressing mode

The mnemonic you typed doesn't work with the addressing modes you specified. Check your 68000 manual for allowable combinations.

invalid addressing mode

The combination of addressing modes you picked for the **movem** instruction are not implemented by the 68000. Check your 68000 reference manual for details.

invalid symbol following ^^

What followed the ^^ wasn't a valid symbol at all.

mis-nested .endr

The assembler found a .endr directive when it wasn't prepared to find one. Check your repeat-block nesting.

mismatched .else

The assembler found a .else directive when it wasn't prepared to find one. Check your conditional assembly nesting.

mismatched .endif

The assembler found a .endif directive when it wasn't prepared to find one. Check your conditional assembly nesting.

missing '='**missing '}'****missing argument name****missing close parenthesis ')'****missing close parenthesis ']'****missing comma****missing filename****missing string****missing symbol****missing symbol or string**

The assembler expected to see a symbol/filename/string (etc...), but found something else instead. In most cases the problem should be obvious.

misuse of '.', not allowed in symbols

You tried to use a dot (.) in the middle of a symbol name.

mod (%) by zero

The expression you typed involves a modulo by zero.

Error Messages

multiple formal argument definition

The list of formal parameter names you supplied for a macro definition includes two identical names.

multiple macro definition

You tried to define a macro which already had a definition.

non-absolute byte reference

You tried to make a byte reference to a relocatable value, which the object file format does not allow.

non-absolute byte value

You tried to dc.b or dcb.b a relocatable value. Byte relocatable values are not permitted by the object file format.

register list order

You tried to specify a register list like D7-D0, which is illegal. Remember that the first register number must be less than or equal to the second register number.

register list syntax

You made an error in specifying a register list for a .reg directive or a .movem instruction.

symbol list syntax

You probably forgot a comma between the names of two symbols in a symbol list, or you left a comma dangling on the end of the line.

syntax error

This is a "catch-all" error.

undefined expression

The expression has an undefined value because of a forward reference, or an undefined or external symbol.

unimplemented addressing mode

You tried to use 68020 "square-bracket" notation for a 68020 addressing mode. MADMAC does not support 68020 addressing modes.

unimplemented directive

You have found a directive that didn't appear in the documentation. It doesn't work.

unimplemented mnemonic

You've found an assembler (or documentation) bug.

unknown symbol following ^^

You followed a ^^ with something other than one of the names defined, referenced or streq.

unsupported 68020 addressing mode

The assembler saw a 68020-type addressing mode. MADMAC does not assemble code for the 68020 or 68010.

unterminated string

You specified a string starting with a single or double quote, but forgot to type the closing quote.

write error

The assembler had a problem writing an object file. This is usually caused by a full disk, or a bad sector on the media.

